[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 7:Transactions

Posted by CHARLES *on* 2020-04-12

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.

—James Corbett et al., **Spanner: Google's Globally-Distributed Database** (2012)

- A **transaction** is a way for an application to group several reads and writes together into a logical unit.
  - Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (commit) or it fails (abort, rollback).
- By using transactions, the application is free to ignore certain potential error scenarios and concurrency issues, because the database takes care of them instead (we call these safety guarantees)
  - vs. **higher performance** or **higher availability**
- **How do you figure out whether you need transactions** ?
  - In order to answer that question, we first need to understand exactly what safety guarantees transactions can provide, and what costs are associated with them.

The Slippery Concept of a Transaction

- Almost all relational databases today, and some non-relational databases, support transactions. E.g. MySQL, MSSQL, Oracle, PostgreSQL etc.

and Durability.

- Systems that do not meet the ACID criteria are sometimes called **BASE**, which stands for **Basically Available, Soft state, and Eventual consistency**.
  - This is even more vague than the definition of ACID.
- **Atomicity**: atomic refers to something that cannot be broken down into smaller parts
  - If the writes are grouped together into an **atomic transaction**, and the transaction cannot be completed (committed) due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction.
- **Consistency**: refers to an application-specific notion of the database being in a "good state."
  - You have certain statements about your data (invariants) that must always be true—e.g. In an accounting system, credits and debits across all accounts must always be balanced.
    - However, this depends on the application's notion of invariants.
  - **Atomicity, isolation, and durability** are properties of the database, whereas **consistency** (in the ACID sense) is a property of the application.
- **Isolation**: concurrently executing transactions are isolated from each other: they cannot step on each other's toes. (old term: Serializability)
  - In practice, serializable isolation is rarely used, because it carries a performance penalty.
- **Durability**: the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.
  - **In a single-node database**, durability typically means that the data has been written to non-volatile storage such as a hard drive or SSD.
  - **In a replicated database**, durability may mean that the data has been successfully copied to some number of nodes.
  - perfect durability does not exist. (e.g. What if earth were destroyed)
- **Replication and Durability**:
  - In practice, there is no one technique that can provide absolute guarantees. There are only various risk-reduction techniques, including writing to disk, replicating to remote machines, and backups—and they can and should be used together.

- **Single-Object and Multi-Object Operations**:
  - Multi-object transactions are often needed if several pieces of data need to be kept in sync. (e.g. Email app, with "unread" counter, a way of denormalization)
  - **Violating isolation**: one transaction reads another transaction's uncommitted writes (a "dirty read").
  - **Multi-object transactions** require some way of determining which read and write operations belong to the same transaction.

    - Increment operation, which removes the need for a read-modify-write cycle.
    - **Compare-and-set** and other single-object operations have been dubbed "light-weight transactions" or even "ACID" for marketing purposes
  - **The need for multi-object transactions**:
    - Many distributed datastores have abandoned multi-object transactions because they are difficult to implement across partitions, and they can get in the way in some scenarios where very high availability or performance is required.
    - In a relational data model, a row in one table often has a foreign key reference to a row in another table.
    - In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object
    - In databases with secondary indexes (almost everything except pure key-value stores), the indexes also need to be updated every time you change a value.
- **Handling errors and aborts**:
  - A key feature of a transaction is that it can be aborted and safely retried if an error occurred.
    - if the database is in danger of violating its guarantee of atomicity, isolation, or durability, it would rather abandon the transaction entirely than allow it to remain half-finished.
    - But datastores with leaderless replication won't obey such rules but work much more on a "best effort" basis, so it's the application's responsibility to recover from errors.
  - Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn't perfect.

Weak Isolation Levels

- If two transactions don't touch the same data, they can safely be run in parallel, because neither depends on the other.
- databases have long tried to hide concurrency issues from application developers by providing **transaction isolation**.
  - **serializable isolation** means that the database guarantees that transactions have the same effect as if they ran serially (i.e., one at a time, without any concurrency)
- **Serializable isolation** has a performance cost, and many databases don't want to pay that price
  - therefore common for systems to use weaker levels of isolation, which protect against some concurrency issues, but not all.
- Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them.
- **Several weak (non-serializable) isolation levels that are used in practice**:

- **Write Skew and Phantoms;**

- **Read Committed**: The most basic level of transaction isolation with two guarantees(default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, and many other databases)
  - 1. When reading from the database, you will only see data that has been committed (no dirty reads).
  - 2. When writing to the database, you will only overwrite data that has been committed (no dirty writes).
  - **No dirty reads**:
    - Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a dirty read;
  - **No dirty writes**:
    - what happens if the earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value? This is called a dirty write.
    - by delaying the second write until the first write's transaction has committed or aborted.
  - **Implementing read committed**:
    - Databases **prevent dirty writes** by using **row-level locks**: when a transaction wants to modify a particular object (row or document), it must first acquire a lock on that object.
    - Most databases **prevent dirty reads** using the approach: for every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock.
- **Snapshot Isolation and Repeatable Read**: (it is supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others)
  - Snapshot isolation is a popular feature;
  - **Non-repeatable read or read skew**: if Alice were to read the balance of account 1 again at the end of the transaction, she would see a different value ($600) than she saw in her previous query.
  - **Snapshot isolation** is the most common solution to this problem. The idea is that each transaction reads from a **consistent snapshot** of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction.(C: similar with Version Vector?)
    - Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.
  - **Snapshot isolation** is a boon for long-running, read-only queries such as backups and analytics.
  - **Implementing snapshot isolation**:
    - Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes;

    - From a performance point of view, a key principle of snapshot isolation is readers never block writers, and writers never block readers.
    - Because it maintains several versions of an object side by side, this technique is known as **Multi-Version Concurrency Control (MVCC)**.
  - **Visibility rules for observing a consistent snapshot**:
    - When a transaction reads from the database, transaction IDs are used to decide which objects it can see and which are invisible.
    - an object is visible if both of the following conditions are true:
      - At the time when the reader's transaction started, the transaction that created the object had already committed.
      - The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.
    - By never updating values in place but instead creating a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead
    - **Indexes and snapshot isolation**: (?)
      - append-only/copy-on-write variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page.
  - **Repeatable read and naming confusion**:
    - Many databases that implement it call it by different names. In Oracle it is called serializable, and in PostgreSQL and MySQL it is called repeatable read.
    - It defines repeatable read, which looks superficially similar to snapshot isolation.(C: Because the concept of "snapshot-isolation" haven't invented yet back then)
    - As a result, nobody really knows what repeatable read means.
- **Preventing Lost Updates**:
  - There are several other interesting kinds of conflicts that can occur between concurrently writing transactions. The best known of these is the **lost update** problem.
  - **The lost update problem can occur** if an application reads some value from the database, modifies it, and writes back the modified value (a read-modify-write cycle).
    - If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification.
    - E.g.
      - 1, Incrementing a counter or updating an account balance;
      - 2, Making a local change to a complex value;
      - 3, Two users editing a wiki page at the same time
  - **Atomic write operations**: (e.g. MongoDB, Redis)

example, updates to a wiki page involve arbitrary text editing

- **cursor stability**: Atomic operations are usually implemented by taking an exclusive lock on the object when it is read so that no other transaction can read it until the update has been applied.
  - Another option is to simply force all atomic operations to be executed on a single thread.
- **Explicit locking**:
  - if the database's built-in atomic operations don't provide the necessary functionality, it is for the application to explicitly lock objects that are going to be updated.
    - E.g. The **FOR UPDATE** clause indicates that the database should take a lock on all rows returned by this query.
- **Automatically detecting lost updates**: (e.g. PostgreSQL, Oracle, SQL)
  - An alternative is to allow them to execute in parallel and, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.
- **Compare-and-set**:
  - In databases that don't provide transactions, you sometimes find an atomic **compare-and-set** operation.
    - avoid lost updates by allowing an update to happen only if the value has not changed since you last read it.
- **Conflict resolution and replication**:
  - Locks and compare-and-set operations assume that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context.
  - **Solution**: allow concurrent writes to create several conflicting versions of a value (also known as siblings), and to use application code or special data structures to resolve and merge these versions after the fact.
  - The last write wins (LWW) conflict resolution method is prone to lost updates and Unfortunately, LWW is the default in many replicated databases.

- **Write Skew and Phantoms**: (e.g. Hospital doctor on call schedule App)
  - **Write skew**. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice's and Bob's on-call schedule, respectively).
    - You can think of Write skew as a generalization of the lost update problem.
  - **Write skew** can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects).
    - Automatically preventing write skew requires true serializable isolation.
    - The second-best option in this case is probably to explicitly lock the rows that the transaction depends on.
  - **More examples of write skew**:

because Email usually guarantee to be unique) Unique constraint could solve this problem;
    - Preventing double-spending;
- **Phantoms causing write skew**:
    - where a write in one transaction changes the result of a search query in another transaction, is called a **phantom**.
- **Materializing conflicts**: (last resort, A serializable isolation level is much preferable in most cases.)
    - If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?
    - This approach is called **materializing conflicts**, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database.

Serializability

- **Serializable isolation** is usually regarded as the strongest isolation level.
    - It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency.
- Three techniques of implementation: (Focus one node in this chapter)
    - Literally executing transactions in a **Serial-order**;
    - **Two-phase locking**, which for several decades was the only viable option;
    - **Optimistic concurrency control** techniques such as **serializable snapshot isolation**;
- **Actual Serial Execution**: (e.g. Redis, VoltDB/H-Store, Datomic)
    - The simplest way of avoiding concurrency problems is to **remove the concurrency entirely**: to execute only one transaction at a time, in serial order, on a single thread.
    - Two Reasons why this starting pick momentum late until 2007:
        - RAM became cheap enough that for many use cases is now feasible to keep the entire active dataset in memory.
        - Database designers realized that OLTP transactions are usually short and only make a small number of reads and writes
    - Sometimes perform better than a system that supports concurrency, because it can avoid the coordination overhead of locking, but its throughput is limited to that of a single CPU core.
    - **Encapsulating transactions in stored procedures**:
        - systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions.
            - Instead, the application must submit the entire transaction code to the database ahead of time, as a **stored procedure**.
        - **Pros and cons of stored procedures**:

        - Code running in a database is difficult to manage: compared to an application server;
        - A database is often much more performance-sensitive than an application server, because a single database instance is often shared by many application servers. (C: wider "negative" impact if a badly written SP)
    - Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead: VoltDB uses Java or Groovy, Datomic uses Java or Clojure, and Redis uses Lua.
- **Partitioning**:
    - For applications with high write throughput, the single-threaded transaction processor can become a serious bottleneck.
    - If you can find a way of partitioning your dataset so that each transaction only needs to read and write data within a single partition, then each partition can have its own transaction processing thread running independently from the others.
    - Whether transactions can be single-partition depends very much on the structure of the data used by the application.
- **Summary of serial execution**:
    - a viable way of achieving serializable isolation within certain constraints:
        - Every transaction must be small and fast, because it takes only one slow transaction to stall all transaction processing.
        - It is limited to use cases where the active dataset can fit in memory. (C: use anti-caching if the dataset is in disk)
        - Write throughput must be low enough to be handled on a single CPU core or transactions need to be partitioned without requiring cross-partition coordination.
        - Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.
- **Two-Phase Locking (2PL)**: (2PL is not 2PC)
    - Several transactions are allowed to concurrently read the same object as long as nobody is writing to it. But as soon as anyone wants to write (modify or delete) an object, exclusive access is required:
        - If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue. (This ensures that B can't change the object unexpectedly behind A's back.)
        - If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue. (Reading an old version of the object, like in Figure 7-1, is not acceptable under 2PL.)
    - In 2PL, writers don't just **block other writers; they also block readers** and vice versa.

    - Shared and Exclusive Lock;
    - The first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.
    - Since so many locks are in use, it can happen quite easily that transaction A is stuck waiting for transaction B to release its lock, and vice versa. → **deadlock**
        - The database automatically detects deadlocks between transactions and aborts one of them so that the others can make progress.
        - Application need retry after abort
- **Performance of two-phase locking**:
    - transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.
    - databases running 2PL can have quite unstable latencies, and they can be very slow at high percentiles
- **Predicate locks**:
    - A database with serializable isolation must prevent phantoms.
    - **Predicate lock**: It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition.
    - A predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms).
- **Index-range locks**:
    - Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming.
    - Most databases with 2PL actually implement **index-range locking** (also known as next-key locking), which is a simplified approximation of predicate locking.
    - Index-range locks are not as precise as predicate locks would be, but since they have much lower overheads, they are a good compromise.
- **Serializable Snapshot Isolation (SSI)**: (new default in the future)
    - It provides full **serializability**, but has only a small performance penalty compared to **snapshot isolation**.
        - SSI is fairly new: it was first described in 2008 and is the subject of Michael Cahill's PhD thesis;
    - **Pessimistic vs. Optimistic concurrency control**:
        - Two-phase locking is a so-called **pessimistic** concurrency control mechanism;
            - Serial execution is, in a sense, pessimistic to the extreme.
        - Serializable snapshot isolation is an **optimistic** concurrency control technique
    - Contention can be reduced with commutative atomic operations

serialization conflicts among writes and determining which transactions to abort.

- **Decisions based on an outdated premise**:
  - How does the database know if a query result might have changed? There are two cases to consider:
    - **Detecting reads** of a stale MVCC object version (uncommitted write occurred before the read)
    - **Detecting writes** that affect prior reads (the write occurs after the read)
- **Detecting stale MVCC reads**:
  - In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules.

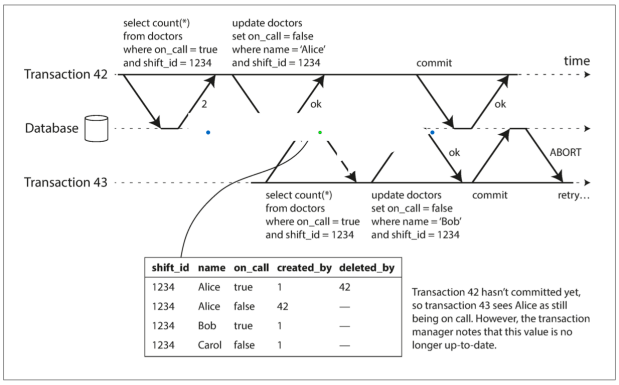

Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

  - By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.
- **Detecting writes that affect prior reads**:
  - The second case to consider is when another transaction modifies data after it has been read.

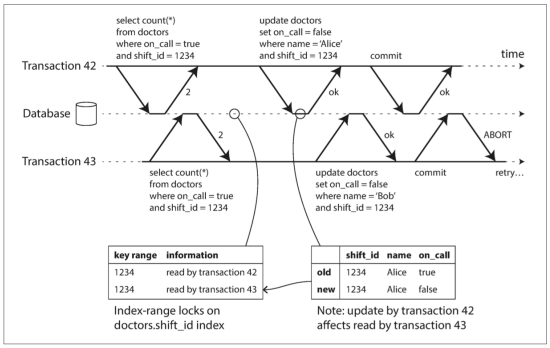

Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

  - When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data.
    - This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a **tripwire**: it simply notifies the transactions that the data they read may no longer be up to date.

by another transaction.
    - Like under snapshot isolation, writers don't block readers, and vice versa.
- The rate of aborts significantly affects the overall performance of SSI.

Summary

- Transactions are an abstraction layer that allows an application to pretend that certain concurrency problems and certain kinds of hardware and software faults don't exist.
- Concurrency control: (isolation levels)
    - Read committed,
    - Snapshot isolation (sometimes called Repeatable read),
    - Serializable.
- Race conditions:
    - Dirty reads
    - Dirty writes
    - Read skew (non-repeatable reads)
        - Solved by: snapshot isolation which implemented with MVCC;
    - Lost updates
    - Write skew
        - Solved by: serializable isolation
    - Phantom reads
        - Solved by: index-range locks
- Only Serializable Isolation protects against all of these issues. We discussed three different approaches to implementing serializable transactions:
    - Literally executing transactions in a serial order
    - Two-phase locking
    - Serializable snapshot isolation (**SSI**) (C:best approach)

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Designing Data Intense Application – Chapter 6: Partitioning

NEXT POST
Designing Data Intense Application – Chapter 8: The Trouble with Distributed Systems